

Replica placement for p2p redundant data storage on unreliable, non-dedicated machines

Abstract—P2P architecture appears to fit for enterprise backup. In contrast to dedicated backup servers, nowadays a standard solution, making backups directly on organization's workstations should be cheaper (as existing hardware is used) and more efficient (as there is no single bottleneck server). However, non-dedicated machines cause other challenges. Update propagation algorithms must take into account frequent transient failures (computers turned off for nights or weekends). Hardware and network topology are heterogeneous. A replication system must not only handle resulting constraints (such as e.g. varying amount of disc space), but also take advantage of the diversity (and e.g., distribute replicas geographically).

We present a p2p backup system using pairwise replication contracts between a data owner and a replicator (storing one of the replicas). In contrast to storing the data directly in a DHT, the contracts allow the system to optimize replicas' placement depending on features of the data owner, the replicators and the network topology. To cope with replicators' transient failures, the system propagates notification about the updates through asynchronous messages. Each peer has a few *synchro-peers* that, in case the peer is off-line, store these notifications. Once back online, the peer contacts its synchro-peers, and then explicitly requests latest versions of modified files it replicates from the owner, or from other replicas.

We implemented the system and tested it on PlanetLab as well as 150 computers in the faculty's student computer labs. We experimentally assess the *cost of unavailability* showing how the time needed for data backup increases non-linearly with machines' unavailability.

Individual workstations are certainly less reliable and less efficient than a dedicated backup server. Yet, as our experiments show, given a sufficient number of such workstations, a p2p backup system outperforms a centralized solution, benefiting from better scalability and distributed topology. P2P is a cheap and robust alternative for enterprise backup.

Index Terms—distributed storage, enterprise backup, data replication, unstructured p2p networks, availability

I. INTRODUCTION

Large corporations, medium and small enterprises, and common computer users are all interested in protecting their data against hardware failures. Currently, replicas of the data are most often maintained at dedicated *backup servers*. Such approach suffers from many issues. Firstly, replicating all user data to ensure satisfactory resiliency level requires significant storage space — which is expensive. Moreover, with a large number of workstations that must be replicated at backup servers, the servers become a bottleneck and may be not able to offer satisfactory throughput. In addition, even if there is additional replication between the backup servers, unless the servers are geographically scattered, the data is not well protected against disasters (e.g. fire, flood). Similarly, keeping all the data at a single place is unsafe in terms of privacy and

security against theft. In this paper, we explore an alternative for traditional secondary storage: a system for data replication based on the p2p paradigm.

According to [1], a large fraction of disk space on desktop workstations is unused. Moreover, this fraction is increasing every year. Motivated with this observation we built a scientific prototype of the system which replicates user data between different workstations of the same network. We assume that the workstations are heterogeneous and prone to failures, in particular: (i) hardware might be heterogeneous and inefficient; (ii) the workstations may have variable amount of unused disk space (the space that is available for keeping replicas); (iii) the workstations are not always available — computers might stay powered on, or be powered off, when not used by anyone (iv) they may experience permanent failures after which it is not possible to recover data stored on a machine. We assume that a community interested in protecting its data (e.g. a single enterprise, an organization, or a group of friends) connects its machines to our backup system. Once in the system, besides the standard activities, machines also keep replicas of data of other peers. Except for installing software, there is no additional administrative overhead and (almost no) additional energy consumption — as the backup system opportunistically uses machines' on-line time for transferring replicas. The next few paragraphs present the issues we had to face when designing and implementing the prototype.

Data storage in our system is based on storage contracts between an owner of the data and its replicas. A peer who wants to replicate its data must firstly sign a contract with another peer that will become a replica. In contrast with the usual approach of storing the data in a Distributed Hash Table (DHT) [2]–[6], the storage contracts allow for any chunk of data to be placed at any location, depending on the optimization goal. Thus, the contracts form an unstructured, decentralized architecture that enables to optimize replica placement, making the system both more robust and able to take advantage of network and hardware configuration. Additionally, contracts allow strategies for replica placement that are incentive-compatible, such as mutual storage contracts [7], [8]. However, we do use a DHT to store a limited amount of metadata. Section III details the architecture of the system.

As our system runs on non-dedicated hardware, the workstations may be unavailable for some time just because they are temporarily powered off. In contrast to many distributed storage systems (e.g., GFS [9]), in such a case our system does not rebuild the missing replica immediately, in order not to generate unnecessarily load on other machines nor the

network. Instead, when the unavailable peer eventually joins back the network, it efficiently updates its replicas. Once back online, such a peer contacts its *synchro-peers* — who keep the messages with the current versions of data chunks — processes the messages and then gets the updated data from the owner, or other replicas (Section IV details the mechanism). Naturally, after a long period of unavailability, the peer is considered permanently damaged and the replicas it was storing are rebuilt on different machines.

Because peers have heterogeneous characteristics and different amount of space destined to keep other peers' replicas, there is a need for a mechanism managing the placement of the replicas (Section V). This mechanism is responsible for relocating or additionally replicating portions of data that do not achieve sufficient resiliency level, either because the number of replicas is insufficient, or the geographic distribution of the replicas is unsatisfactory. In order to protect the data against natural disasters, at least one replica should be placed at a geographically remote location. However, in case of more common, small-scale local failures, to make the data available for fast retrieval, there should be also a replica close to the data owner. The distributed protocol of replicas placement optimization relies on gossiping of peers' resiliency information. Peers build a distributed priority queue that contains the least replicated data. Peers who have some free storage space offer it to the least-replicated peers.

We have implemented a prototype system and tested it on 150 computers in students' computer laboratories; and on 50 machines in Planet-Lab. The lab environment might be considered as a worst-case scenario for an enterprise network, as the computers have just 13% average availability and are frequently rebooted. Moreover, we assumed that all the local data is modified daily. The results of our experiments show that: (i) even with very low availabilities of the machines we are able to efficiently balance the load on the machines (ii) in a p2p architecture we are able to efficiently transfer data chunks — the bandwidth of such a system scales linearly with the number of machines; (iii) we also show that backup duration is significantly influenced by the unavailability of the machines (we call this effect the *cost of unavailability*); (iv) because of unavailability the time needed for direct communication of two peers can be long (on average 20h); when we do not require synchronization of the communicating peers the asynchronous messages efficiently reduce the receiver waiting time for a message delivery.

Our prototype implementation (with the source code) is available for download at <http://dl.dropbox.com/u/70965741/p2pBackup.tar>. We plan to release it with an open-source license.

The main contribution of our paper is an architecture of a p2p backup system that is based on unstructured, decentralized storage contracts between the data owner and the replicas. We validate this architecture through a prototype implementation and a series of experiments (Section VI). Besides (i) a new architecture with storage contracts, our work has the following contributions. (ii) We propose asynchronous messaging

mechanism, based on synchro-peers, suitable for managing replicas in case of frequent transient failures (Section IV). (iii) We propose a distributed mechanism to assess resiliency level and to balance the load and the resiliency levels of replicas according to the architectural diversification of the workstations (Section V).

II. RELATED WORK

Data replication in distributed systems has been addressed by both scientific and commercial projects. In this section we analyze the related theory and the contribution of the real systems offering data replication.

HYDRAsTOR [10] and Data Domain [11] are commercial distributed storage systems, which use data deduplication [12] to increase amount of the virtual disk space. Although these systems are scalable and offer high performance, they use specially dedicated servers and therefore are expensive to get and to properly maintain (see the Introduction for more arguments). Because they are just storage systems (in contrast to file systems), they also require properly configured backup applications and network to achieve high throughput.

Many papers analyze various aspects of p2p storage by either simulation or mathematical modeling. Usually, the analysis focuses on probabilistic analysis of data availability in presence of peers' failures (e.g., [13]). In [14], similarly to our system, the goal is to optimize availability of a set of files over a pool of hosts with given availability: theoretical as well as simulation results are provided for file availability. [15] studies by simulation durability and availability in a large scale storage system. [16] and [17] show basic analytical models and simulation results for data availability under replication and erasure coding. Our paper complements these works by, firstly, making experiments also on a prototype (and not — only on a model); and, secondly, by considering other measures of efficiency (like time of data access, update propagation, etc.).

As the context of this work is data backup in a single organization, we do not analyze incentives to participate in the system. However, to store the data, our system relies on agreements (contracts) between peers. In contrast, in DHT-based storage systems contracts are reached between a peer and a system as a whole. Thus, our system naturally supports methods of organization that emphasize incentives for high availability, such as mutual storage contracts [7], [8].

Many p2p file systems [2]–[6], use storage and routing based on a DHT [18], [19]. The address of the block, which is a hash of its content, fully determines the locations of its replicas. Thus, it is neither possible to balance the load on replicating workstations, nor optimize placement of replicas (besides balancing the DHT). While these solutions focus on ensuring consistency of the data being modified by multiple users, this paper focuses on the issue of the best replication of the data, which cannot be modified by anyone apart from its owner.

OceanStore [20] and Cleversafe [21] propose the idea of spreading replicas into geographically remote locations achieving the effect of a deep archival storage. These systems com-

bine software solutions with a specially designed infrastructure that consists of numerous, geographically-distributed servers. The main contribution of these systems, from our perspective, is the resignation from a common DHT and the introduction of a new assumption that any piece of data can be possibly located at any server. These systems, however, do not discuss the issue of replicating data on the ordinary workstations (which are, in contrast to the servers, frequently leaving and joining the network) and do not present any means allowing to handle such dynamism.

Wuala [22] moved one step further by proposing a distributed storage based not only on a specially dedicated infrastructure, but also including a cloud of workstations of users who install Wuala application. However, since late 2011, Wuala no longer supports p2p storage. Other p2p backup software include Backup P2P [23], Zoogmo [24], or ColonyFs [25]. To our knowledge, there are no papers describing these systems, so a proper comparison cannot be made.

FreeNet [26] is a p2p application that exposes the interface of a file system. Its main design requirement is to ensure anonymity of both authors and readers. The underlying protocol relies on proximity-based caching. When a data item is no longer used, it can be removed from a caching location. From FreeNet, we took the idea of placing at least one replica close to the data owner in order to achieve fast data retrieval. However, we extended it into a more generic utility function that takes into account more properties.

Farsite [27] was a Microsoft's 6-years long project aimed at creating distributed file system for sharing data between thousands of users. [27] gave us the feeling of following a good direction. Firstly, it emphasizes that real scalability must face the problem of constant failures in the network. Our work implements asynchronous messages that enable communication between unavailable peers. Secondly, Farsite creators claim that in a scalable system, manual administration must not increase with the size of the network. In our system, all participants have the same role and none of them requires any additional configuration.

There are a few substantial differences between Farsite and our work. Most importantly, Farsite's architecture does not rely on mutual contracts, which allow us to implement both incentives and mechanisms ignoring the black listed peers.

In Farsite updates of data are committed locally and the changes are appended to the log (similarly to Coda [28]). The log is sent to a group of peers responsible for managing a subset of a global name space (called directory group), which periodically broadcast log to the all group members informing the interested replicators about the changes in data. Because directory group uses Byzantine Fault Tolerant protocol [29] no file can be modified if one third or more of the group members is faulty. Because we consider a backup system in which data is modified only by the owner (differently from Farsite) we are able to gain in flexibility and robustness. In our asynchronous updates mechanism, every peer has associated group of synchro-peers managing its asynchronous messages. Synchro-peers are independent of replicas, which results in

a desired property that every peer can keep replicas for any chunk of data. Therefore, replicas can be chosen so that they constitute the most profitable replication group. Moreover, updates can be propagated if only one synchro-peer is on-line, which increases system robustness.

Finally, in Farsite the replicating workstations are chosen randomly and the background process swaps machines replicating data in order to maximize the minimal data availability [30]. We propose more generic utility function that takes into account many factors (data availability is only one of them). Replica placement (scored through the utility function) is optimized by an efficient distributed protocol. Differently from Farsite's queuing, throttling, shedding, clown car, etc. [31], smart placements of the replicas allows us to optimize the placement and to balance the load between workstations.

III. SYSTEM ARCHITECTURE

Our system uses a mixed architecture that stores differently meta-data and data. Meta-data consists of basic control information that allows peers to locate and to connect to each other. Meta-data is small, but must be located efficiently — hence we use a DHT as a storage mechanism. In contrast, each peer is responsible for finding and managing peers who replicate its data (its *replicas*). Such replication contracts enable us to optimize replica placement and thus to tune replication to a specific network configuration.

A. Assumptions, Notation and Vocabulary

Our system consist of *peers* (denoted by i, j, \dots), also called *workstations* or *machines*. Each peer is uniquely identified by an ID and has a pair of keys for secure, asymmetric communication.

Our system stores *chunks of data* denoted by $d_k, d_l \dots$ (or $k, l \dots$ when it is unambiguous). Each chunk of data d_k has a single peer who owns it — an *owner* $o(d_k)$. Our system replicates encrypted d_k on other peers, denoted as *replicas* $R_k = R(d_k) = \{j\}$ (the system guarantees that $o(d_k) \notin R(d_k)$). By $P_k = P(d_k)$ we denote *data placement*, a tuple $P_k = P(d_k) = (o(d_k), R(d_k))$ (we assume that the owner also stores a local copy of a data chunk).

B. Meta-data

Control information, such as identification and connectivity information is stored in a highly replicated DHT. The DHT allows us to efficiently locate the information. At the same time, compared with normal data, metadata is small. Thus, a DHT can be maintained by the peers without large overhead.

The basic attributes of a peer are kept in a structure called *PeerDescriptor*. Each peer i in our system is uniquely identified by its ID (computed as a hash of its public key). For each peer, its PeerDescriptor contains:

- identification information (public key);
- information needed to connect to this peer (a current IP address and a port of an instance of our software running on a workstation) and account name in the operating system

(account name is required by the current implementation of data transmission layer — see Section III-C3);

- list of the presence timestamps allowing to assess its availability;
- the total size of data this peer replicates;
- identifiers (IDs) of *synchro-peers* who deliver asynchronous messages (Section IV).

PeerDescriptors of all peers are kept in a DHT: peer's ID is hashed to its PeerDescriptor. As the size of the metadata is small, we are able to afford strong replication (compared to a generic DHT). Thus, instead of a single peer, a number of peers is responsible for keeping data hashed to a part of the key space.

C. Data replication

As one of the prototype's main goals is to intelligently place the replicas in the network of peers, we need to be able to store any replica at any peer. This architecture contrasts with content addressable storage systems that put each chunk of data under the address which is fully determined by the chunk's unique identifier (e.g. hash of its content). As a trade-off for flexibility of placing replicas at any location, we need a mechanism to locate data.

1) *Data Catalog*: Each peer keeps information about replica placement of its files in an index structure called *DataCatalog*. This catalog is inspired by a notion of a directory in a standard file system (although it is not recursive — each peer has only one catalog). For each data chunk, the catalog stores information about:

- identifiers of the peers that keep replicas of the data chunk (hereinafter *chunk replicators*);
- size of the chunk;
- version number of the chunk.

The DataCatalog is persisted in a file, however it is not replicated between peers, as any other ordinary file. In that way we avoid the additional overhead of updating it at the remote locations, when the contract for any chunk is changed; because machines are unreliable, in many cases we even would not be able to update DataCatalog at the remote peers as they can be simply unavailable. On the other hand, each peer is aware of its all storage contracts so when the local data of any peer is lost, the information about the failure is gossiped and the DataCatalog can be rebuilt from the other peers' information about their contracts. Once DataCatalog is reconstructed, peer can locate and rebuild any missing data.

2) *Contracts negotiation*: Every member of the network, before placing its replicas at a remote peer, must obtain this peer's permission. The process of reaching the agreement is called the negotiation of the storage contract. Isolating the functionality of the contracts negotiation gives a flexibility of implementing the incentive-compatible replica placement mechanisms, such as mutual storage contracts [7], [8].

Because the context of this paper is a single organization of well-behaving member workstations, the implementation assumes that there are no malicious users acting selfishly

against organization interest. Thus, in the current implementation, contract negotiation is greatly simplified. Every peer agrees to admit any data, if it has sufficient disk space.

After signing the contract, the new replicator authorizes the owner to connect and to transfer the data by coping the owner's public key from the DHT to the replica's key store.

3) *Data Transfers*: The data is transmitted in an encrypted connection. Prior to transmission, peers mutually authorize each other using identities (public keys) available in peers' *PeerDescriptors* (stored in the DHT).

In the current implementation, we use standard Linux tools for data transfers. Each peer runs a ssh daemon that acts as a server that accepts connections of data owners. When a peer initiates connection to transfer its data, it uses rsync as a client.

IV. ASYNCHRONOUS/OFF-LINE MESSAGING

We consider a specific model of a dynamic p2p network in which peers are frequently unavailable because of temporary failures. Hence, a peer that wants to update its replicas at remote replicators or put replicas of new data at remote replicators using previously negotiated contracts is likely to find some remote location unavailable. A common approach [9], [10] is to rebuild replicas at different locations as soon as the original remote location storing a replica is down. Because of frequent transient failures, peers' unavailability is a common case, and thus such approach would result in unneeded network traffic and machine load. In this section we describe a mechanism of asynchronous messages for handling such situations.

We propose an alternative solution in which each peer periodically sends a message containing the most recent version numbers of its data to the replicators. When peer i wants to pass the version numbers of its data to a currently unavailable peer j , it sends an asynchronous message $m(i \rightarrow j)$ to j . The asynchronous message is delivered by a group of *synchro-peers*. Back online, peer j , having received message $m(i \rightarrow j)$, verifies the versions of replicated data chunks and updates the obsolete chunks by uploading them from the owner or from other replicators.

This protocol has two advantages. First, the asynchronous message is delivered with high probability even when the sender is unavailable. Second, the data may be downloaded concurrently from multiple replicators.

An asynchronous message from i to j is sent to the *synchro-peers* of $j - S(j)$. Synchro-peers is a set of peers, defined for every peer j (j is called in this context a *target peer*) that keep asynchronous messages for j . Synchro-peers of j include j , so every message will be delivered to the target peer by the same means as it is delivered to the other synchro-peers. Each synchro-peer periodically tries to send the asynchronous message to the synchro-peers that have not yet received the message; the IDs of synchro-peers that have not yet received the message are attached to the message (thus, the same message can be delivered multiple times to the same peer).

The consecutive messages between any two peers are versioned with sequential numbers (logical clocks). If any

synchro-peer k has not managed to send a message $m_1(i \rightarrow j)$ to the all requested synchro-peers before receiving a subsequent message $m_2(i \rightarrow j)$ with a higher version number, then the synchro-peer drops $m_1(i \rightarrow j)$, as the new message already contains more recent information. Thus, the number of messages that are waiting for delivery in the whole system is bounded by $N \cdot |R(\cdot)| \cdot |S(\cdot)|$, where N is the number of peers, $|R(\cdot)|$ is the average number of replicators per peer and $|S(\cdot)|$ is the number of synchro-peers per peer.

The target peer executes the commands from the message immediately after the first reception but it remembers for each sender the latest version number of the received message. This information protects against the multiple execution of the orders from a single message.

The mechanism of versioning through asynchronous messages reduce the amount of information replicators keep about the structure of replication contracts. In an alternative solution, replicators synchronize directly between each other. However, this requires replicators to know IDs of all other replicators for each data chunk they store. If this information is stored at replicators, changes in replication contracts require multiple updates; if it is stored as meta-data, the size of the meta-data becomes proportional to the number of data chunks in the system. With asynchronous messages, the owner keeps the information about its replication contracts: updates are simple and meta-data is small.

V. REPLICA PLACEMENT

The goal of a replica placement policy is to find and dynamically adapt the locations of the replicas in response to changing conditions (new peers joining, permanent failures, changing characteristics of existing replicators). Finding possible locations for replicas is not trivial, given that peers differ in availability and amounts of free disk space. Moreover, the replica placement policy should take advantage of peers' heterogeneity (like availability, geographic locations, etc.).

Our policy consists of two main parts. First, a utility function (in short *utility*) scores and compares replica placements. A utility is a function that, for a given data owner and a set of possible replicators returns a score proportional to expected quality of replicating data. Second, a protocol manages replica placement in the network in order to maximize the utility of the currently worst placement (max min optimization).

A. Utility function

A user of a backup application is interested in the resiliency level of her data (defined by the desired number of replicas N_r and their proper geographic distribution); and the time needed to retrieve the data in case the local copy is lost (expressed as the desired data read time $Des(T_r)$). Additionally, a user must be able to backup her data (propagate the local updates to replicas) during the time the user is on-line (expressed as a backup window $Des(T_b)$).

Utility function $U : P_k \rightarrow \mathcal{R}$ is a scoring function mapping a replica placement $P_k = P(d_k) = (o(d_k), R(d_k))$ to its score $u_k = U(P_k)$.

The average duration of data backup to replicator j , $E(T_b, j)$ is estimated by:

$$E(T_b, j) = \frac{\sum_{k:j \in R(d_k)} size(d_k)}{p_{av}(j)B_j} \quad (1)$$

Backup duration is proportional to the congestion on the receiving peer $\sum_{k:j \in R(d_k)} size(d_k)$; and inversely proportional to the bandwidth B_j that peer j dedicates for the background backup activities (B_j is bounded by network and disk bandwidth, but can be further reduced by the user). We use a simplified model that does not explicitly consider the network congestion, but this issue is addressed in the next subsection. Moreover, successful write on j is possible only when j is available (hence $p_{av}(j)$).

Assuming that restore operation has no priority over the backup, the average data restore duration $E(T_r, j)$ is computed in the same way.

The geographical distribution of the data is approximated by TTL values. Most of the replicas should be near the owner to reduce the network usage. $close_{max}$ denotes the desired distance for the “nearby” replicas. However, to cope with geographically correlated disasters, one replica should be far: its distance should be between $remote_{min}$ and $remote_{max}$.

The utility is a sum of utilities expressing geographic distribution, backup time (performance) and the number of replicas (with the former two treated essentially as constraints):

$$U(P_k) = U_{geo}(P_k) - L \cdot ||R(d_k)| - N_r| - M \cdot U_{perf}(P_k), \quad (2)$$

where M and L are (large) scaling factors. L penalizes for insufficient number of replica. M penalizes for backups that cannot be finished within the time window. If the backup cannot be done on time, it means for some data we can give no resiliency guarantees and so, even very good geographic distribution properties are useless.

The backup time penalty $U_{perf}(P_k)$ is the sum over the utilities per replica location:

$$U_{perf}(P_k) = \sum_{j \in R(d_k)} U_{perf}(j)$$

where $U_{perf}(j)$ penalizes for insufficient backup window on j -th replicator:

$$U_{perf}(j) = \min(Des(T_b) - E(T_b, j), 0) + \min(Des(T_r) - E(T_r, j), 0)$$

The geographic utility $U_{geo}(P_k)$ considers both “near” and “far” replicas:

$$U_{geo}(P_k) = \min(0, dist_{TTL}(j_{max}) - remote_{min}) + \min(0, remote_{max} - dist_{TTL}(j_{max})) + \sum_{j \in P_k - \{j_{max}\}} \min(0, close_{max} - dist_{TTL}(j))$$

where j_{max} denotes the most distant location and $dist_{TTL}(j)$ denotes the TTL distance between j and the data owner.

Additionally, in order to limit data movement when $U(P_k)$ differs from $U(P'_k)$ only by small value (in our experiments, 10%) we treat these values as equal.

In an enterprise backup system, we assume that all data chunks are equally valuable. Thus, the utility of the whole system is the utility of the worst placement ($\max \min_k u_k$).

B. Distributed optimization protocol

We considered several approaches for maximizing system utility $\max \min_k u_k$. Probably the most straightforward idea is that each peer is responsible for its own utility u_k . This approach allows to implement game-theoretic strategies [7], [8] that protect each peers' selfish interests. Game-theoretic strategies would give the system extra protection against malicious spammers. However, they have the following drawbacks: (i) every peer has to compete with the other participants; in particular, peers with low availability or bandwidth could never achieve satisfactory replication; (ii) even if contracts for these peers are accepted at the cost of rejecting the contracts of the high utility peers, such frequent contracts rejections will result in protocol inefficiencies. Considering these drawbacks we decided to turn to a proactive approach described below.

Every peer i with free storage space periodically chooses a data chunk d_k with low utility, and proposes a new replication agreement with the data chunk's owner o (peers share information on low utility data chunks in a distributed priority queue). The owner either tentatively adds i to its replication set $R(d_k)$ (if the number of replicas $|R(d_k)|$ is lower than the desired resiliency level N_r); or tentatively replaces $j \in R(d_k)$, one of its current replicas, with i (all possible $j \in R(d_k)$ are tested). If the resulting utility $U(P'_k)$ is higher than the current value $U(P_k)$, the owner o tries to change the contracts (see the next section). When the owner rejects the proposition or when it is unavailable, i puts o in a (temporary) taboo list in order to avoid bothering it later with the same proposition.

As the result of continuous corrections of the replicas placements each peer can end up having replicas of different chunks at different peers. Such a machine has many replicators and their monitoring becomes expensive. However, the monitored information (the availability and the size of replicated data) are gossiped; thus the cost of distribution of information is independent on the number of replicators. On the other hand, storage contracts with multiple peers allow to parallelize data transfers and the cost of replicas rebuilding is amortized.

If every peer proposed storage for the owner of the data with the lowest utility, the owner would get overloaded with storage offers (and the remaining data chunks would be ignored). Therefore, each peer sends a message to o with probability p_p such that $p_p = \frac{T}{N}\alpha$, where N is the estimated number of peers in the network, T is the duration of the period, mentioned before, and α is desired number of messages that peer wants to get in a time unit without being overloaded. Given such probability, the expected value of the number of messages, E_m , the owner of data gets in a time unit is: $E_m = p_p \cdot N \cdot \frac{1}{T} = \alpha$.

The system keeps the data chunks with the lowest utility in a distributed priority queue. In our prototype, we implemented the distributed priority queue by a gossip-based protocol. Each peer keeps a fixed number of data pieces with the lowest priorities. It updates this information with its own data pieces and distributes the information to the randomly chosen peers.

C. Changing replication contracts

When nodes parameters (e.g. availability) change, or when a large number of nodes is added to the system, the contracts are continuously renegotiated. If each such change resulted in data migration, the network and the hosts could easily become overloaded. Therefore the process of changing a contract is more elaborate.

1) *Finding the best replicators:* Below, we describe two aspects of the protocol: revocation of inefficient contracts; and recovery from transient failures.

When peer i offers its storage to data owner o , and if o decides that i should replace one of its existing replicators j (as the resulting value of utility function U is higher), then o has to explicitly revoke the contract with j . Thus, changing location of the data of o , from i to j , requires these three peers being on-line. The example below illustrates why revocation of the contracts cannot be realized asynchronously.

Example 1. Consider peer j storing many data chunks of several owners. From the perspective of each owner, as j is comparably overloaded, any new peer joining the network is a better replicator than j . If the contracts could be revoked asynchronously, all the peers would revoke the contract on j during its unavailability. Now j , having no data, can take over all data stored at some other peer k during k 's unavailability, by offering storage space to the all data owners replicating their data at k . Such situation can repeat indefinitely. Each peer is not aware that j has already revoked some of its contracts and that it is not overloaded any more.

However, if the existing replicator j has low availability, on-line revocation of its contract is also improbable. Thus, an owner can revoke a contract with such a plow-available replica (e.g., the first decile of the population) also through an asynchronous message.

Additionally, because the peers are unreliable, the process of contracts negotiation can break at any point leading to inconsistency of the contracts. Two types of inconsistency are possible: an owner o believes j is its replicator, while j is not aware of such contract; or a peer j believes to be o 's replicator, while o thinks j is not. Contracts negotiation is however idempotent and inconsistent contracts can be easily fixed. Each peer periodically sends messages to its replicators with the believed contracts (and versions of data chunks, which allows the replicators to update the chunks of data which are out of date). Each replicator periodically sends similar information to the appropriate owners. Detected inconsistencies can be resolved either by adopting the owner's state; or by accepting a replication agreement.

2) *Committing contracts and transferring data*: In order to reduce the load on the network, replicators cannot change too often; but to maintain high performance, replicators must eventually follow the negotiated contracts. A *non-committed* contract between an owner and a replicator is negotiated, but no data has been transferred. Contracts are committed periodically. For each data chunk, if there is a new contract (negotiated, but not committed), the contract is committed when the time that passed since the last committed contract for this chunk is large enough (e.g., 24 hours). This guarantees that the data is transferred at most once in each time period (e.g. at most once each 24 hours); but even when (non-committed) contracts change often, data is replicated (as the committed contracts represent a snapshot of utility optimization).

After committing a contract, the owner sends a message to the new replicator that requests data transfer. As soon as the new replicator downloads requested chunk, it sends an acknowledgment to the owner. Finally, the owner notifies the old location to remove the chunk.

VI. EXPERIMENTAL EVALUATION OF THE PROTOTYPE

A. Experimental environment

We performed the experiments in two environments: (i) computers in the faculty's student computer labs; and (ii) PlanetLab. We run the prototype software for over 4 weeks in the labs and 3 weeks in PlanetLab. Each computer acted as a full peer: owned some data and acted as a replicator. The data was considered as modified at the beginning of each day; thus each day we expected the system to perform a complete backup. If the transfer of a particular data chunk did not succeed within a day, the following day we transferred a newer version of the chunk. We used chunks of equal size – 50MB.

The computers were centrally monitored; the central monitoring server experienced several failures which slightly influence the presented results (the real backup times are slightly shorter than presented).

1) *Students computer lab*: We run our prototype software on all 150 machines from the students computer lab. The availability pattern might be considered as a worst case scenario for an enterprise setting. The lab is open from Mondays to Fridays between 8:30am and 8pm and on Saturdays between 9am and 2:30pm. The students frequently (i) switch off or (ii) reboot machines to start Windows; each day at 8pm the computers are (iii) switched off by the administrators (the machines are not automatically powered on the next day); each of these events was considered a transient failure.

The amount of local data was sampled from the distribution of storage space used by the students on their home directories (scaled so that the average value was 3GB); the resulting distribution is similar to uniform between 0 and 8 GB.

The local storage space depended on machines' local hard disks; and varied between 10GB (50% machines), 20GB (10% machines), and 40GB (40% machines).

The computers in students lab have very low average availability (the median is equal to 13%). Figure 1 presents the distribution of the availabilities of the computers in lab.

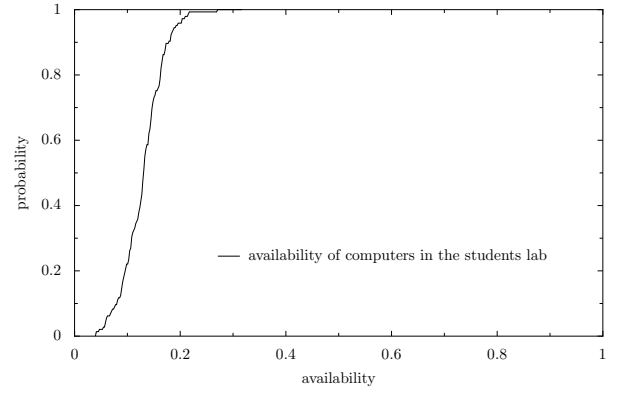


Fig. 1. The cumulative distribution function of the availability of the computers in students lab.

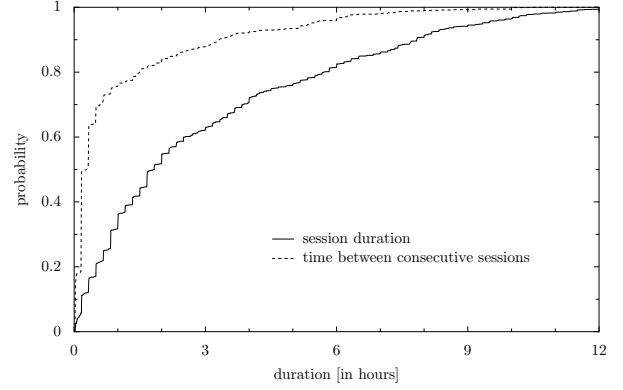


Fig. 2. The cumulative distribution function of the session durations and the times between consecutive sessions for the computers in students lab (the nights are filtered out).

Figure 2 presents the distribution of the up time of the computers and the time between their consecutive availability periods within a single day (the nights are filtered out). Low availability coupled with long session times constitute a worst-case scenario for a backup application: in contrast to short, frequent sessions, here machines are rather switched on for a day, then switched off when the lab closes and remaining off during the next week.

2) *PlanetLab*: The experiments on PlanetLab were using 50 machines scattered around Europe. Each machine was provided 10GB of storage space and had 1GB of local data intended to be backed up. The machines were almost continuously available (availability equal 0.91).

B. Asynchronous messages

In this subsection we present how the asynchronous messaging influence message delivery time and the probability that the message is delivered. For our analyzes we used the traces of availability from the students computer lab. We varied the number of synchro-peers per peer between 0 and 30. For each number of synchro-peers, we generated 100,000 messages with random source, destination and sent time. Figures 3 and 5 present averages and standard deviations.

Figure 3 presents the dependency between the number of synchro-peers and the delivery time of the message. Because the message delivery can be accomplished only when the

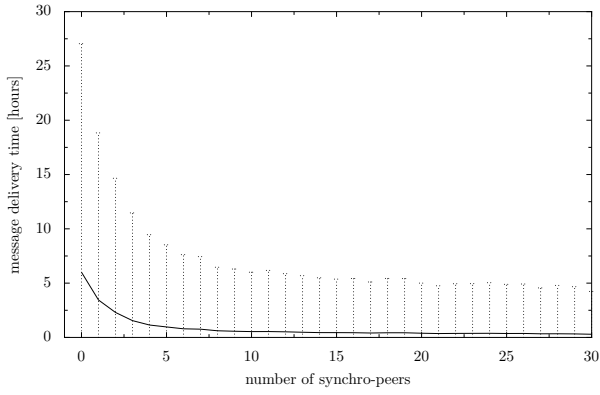


Fig. 3. The dependency between the number of synchro-peers and the delivery time of the asynchronous message. Delivery time measured from the first time of availability of the receiver after the message was sent.

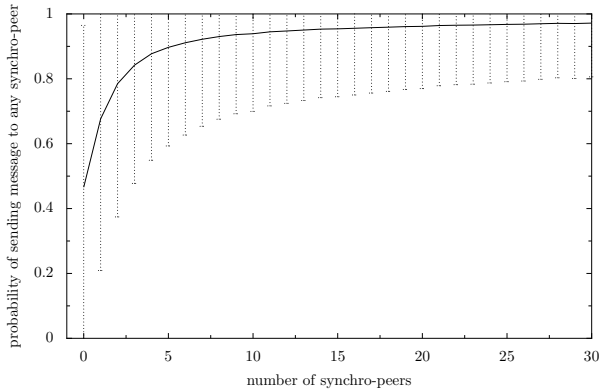


Fig. 4. The dependency between the number of synchro-peers and the probability of delivery a message to any synchro-peer.

receiver is active, we present delivery time measured from the first availability of the receiver after the message was sent. Ideally, the message should be delivered just after the receiver becomes online. The presented result show that the delivery time, measured from the perspective of the receiver, decreases significantly when using synchro-peers. Additionally, the standard deviation, which because of very low peers availabilities is very high, decreases even more. For the number of synchro-peers higher than 5, the advantage of using more of them becomes less significant. Taking into account that the higher number of synchro-peers results in higher number of messages required for synchronization we decided to use 5 synchro-peers for our prototype system.

Figure 5 presents the dependency between the number of synchro-peers and the probability of a successful delivery a message to any synchro-peer. We are interested in calculating such probability because a message delivered to a synchro-peer is, in fact, a replica of the original message. Thus, synchro-peers should enable message delivery even in case of long term absence of the sender (e.g. caused by non-transient failure). The results show that synchro-peers significantly increase this probability – with 5 synchro-peers the system delivers 90% of the messages, while without synchro-peers, more than half of the messages are lost.

TABLE I
THE RATIO: TOTAL SIZE OF REPLICATED DATA (IN MB) TO THE AVAILABILITY FOR THE FIRST 3 DAYS OF EXPERIMENTS IN THE LAB ENVIRONMENT.

day	Utility (weighted replicated data)	
	average	standard deviation
1	34487	6086
2	60489	8141
3	69658	5496

C. Replica placement

1) *Students computer lab*: The goal of tests on the labs was to verify how the system copes with low availability of the machines. For each machine i , we set the bandwidth B_i to the same value and the backup window $Des(T_b)$ to 0. As all the machines are in the same local network, there is no geographical distribution of the data. Thus, the utility function (Eq. 2) degrades to the number of replicas and the backup duration (Eq. 1). This means we wanted to minimize the maximal time required for transferring a data chunk, which means minimizing the load on maximally loaded machine. As the result we expected the machines to be loaded proportionally to their availabilities. By Eq. 1, the load on the machine is proportional to the size of data it replicates; thus for each machines, the total size of replicated data should be proportional to machine's availability. Additionally, storage constraints should influence the amount of data stored.

During the first 3 days of experiments we measured the ratio: total size of data replicated by a peer (in MB) to the peer availability. For each day, we considered only the peers were switched on at least once. We also restricted the measurements only to peers with at least 9GB storage space (that could accommodate, on the average, 3 replicas), to separate the effect of insufficient storage space. The average values and the standard deviations of the ratios for the 3 days are presented in Table I. The standard deviation is low in comparison to the average (they deviations are 18%, 13% and 8% of the corresponding average) which shows that the replicas were distributed according to our expectations.

2) *PlanetLab*: The goal of PlanetLab tests was to verify how our system handles geographic distribution and heterogeneity of the machines. In this environment we required the far replica to have TTL distance from the owner in range $\langle 3, 8 \rangle$ ($remote_{min} = 3$ and $remote_{max} = 8$) and other replicas to be as close to the owner as possible ($close_{max} = 0$). Additionally we set the bandwidth B_i to 500 KB/s for half of the machines, and 1000 KB/s for the other half. We also set the backup window $Des(T_b)$ to 4500s. Each machine had the same amount of local data (1GB); the disk space limit was 4GB. We expected that the low-bandwidth machines will be less loaded than those from the high-bandwidth group. Assuming that machines are continuously available, a low-bandwidth machine should replicate at most 2.25GB; and a high-bandwidth machines at most 4.5GB.

We tested two parameter settings that differed by the weight assigned to geographical distribution of replicas (see Section V-A). For $M = 1$ (which means increasing the backup

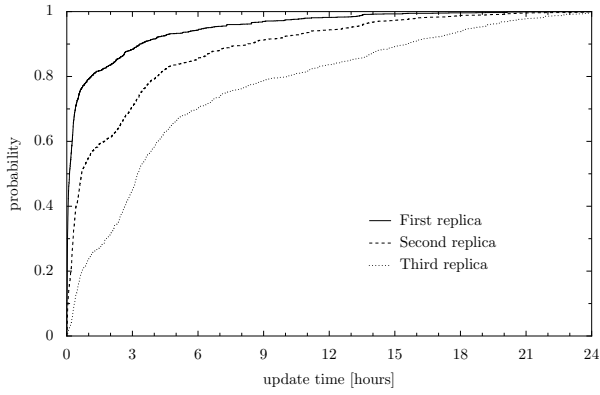


Fig. 5. The cumulative distribution function for the time of creating a replica of data chunk. The time is relative to the data owner. Lab; data collected over 4 weeks of running time.

duration of a single chunk by 1s is equally unwanted as increasing the TTL distance of this chunk by 1), the average TTL distance between the replica and the owner was equal to 11.6 (std dev. 3.7). In this case only two machines exceeded their backup window (by at most 108 s). For $M = 0.01$ (which means increasing the backup duration of a single chunk by 100s is as bad as increasing the TTL distance of a single chunk by 1), the replicas had better geographic distribution: mean TTL equals 8.1 (std dev. 3.8). However, the backup duration was increased – 13 machines exceeded their backup window. The average excess of the backup window was equal to 222s (5% of the backup window) and the maximal 415s (9% of the backup window).

D. Duration of backup of a data chunk

We measured the time needed to achieve the consecutive redundancy levels (the number of replicas) for each data chunk. The time is measured relative to the data chunk owner online time: we multiplied the absolute time by the owner's availability. We consider the relative time as a more fair measure because: (i) the transfer to at least the first replica requires the owner to be available; (ii) data can be modified (and thus, the amount of data for backup grows) only when the owner is available; (iii) we are able to directly compare results from machines having different availabilities.

The distribution of time needed to achieve the consecutive redundancy levels is presented in Figure 5 (lab) and Figure 6 (PlanetLab).

1) *Lab*: The average time of creating the first, the second and the third replica of a chunk are equal to, respectively, 1.1h, 2.7h and 5.5h (the average time needed to create any replica is equal to 3.1h). We consider these values to be satisfactory as the average relative time for transferring a single asynchronous message holding no data (message with 0 synchro-peers), calculated based on availability traces, is equal to 2.6h.

The maximal values, though, are higher: 24h, 29h and 32h. These high durations of replication are almost entirely the consequence of peers' unavailability. The maximal time needed to deliver an asynchronous message with 3 synchro-peers is of the same order (21.5h, measured relatively to source online time, see Section VI-B). Moreover, if we measure only

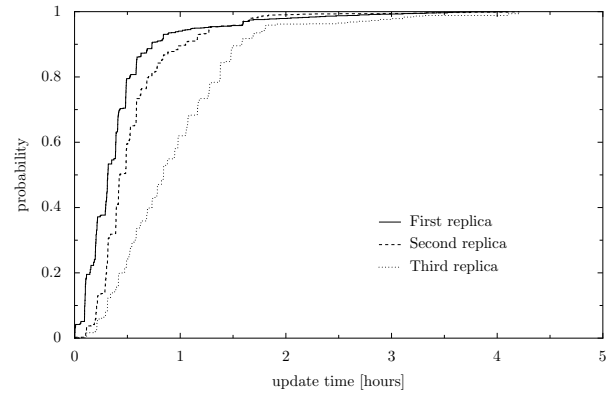


Fig. 6. The cumulative distribution function for the time of creating a replica. The time is relative to the data owner. Planet-Lab; data collected over 3 days.

the nodes with more than 20% average availability, the times needed to create the replicas are equal to 1h, 1.6h and 3h and maximal values are equal to 12h, 18h and 20h.

2) *PlanetLab*: The average times needed for creating the first, the second and the third replica are equal to, respectively, 0.5h, 0.7h and 1.1h. The maximal values are equal to 4.0h, 4.2h, and 4.2h. These values are significantly better than in case of the students computer lab even though the distance between the machines is much higher and the computers in students lab are connected with a fast local network. This once again proves that the unavailability of the machines is the dominating factor influencing the backup duration.

The average time needed for transferring a data chunk is equal to 0.76h. Assuming that the transfer times of chunks are similar, if the chunks are transferred sequentially then the transfer of the half of data is finished within 0.76h. If the chunks are transferred concurrently then almost all the chunks are transferred within 0.76h. Having 1GB of local data and 3 replicas in both cases we can assume that 1.5GB of data is transferred within 0.76h. This gives an estimated throughput of 4.49Mb/s (Planet-Lab uses standard Internet connections).

VII. CONCLUSIONS

We have presented an analysis and a prototype implementation of a p2p backup system based on pair-wise replication contracts. The need for our system came when we realized that although theoretical papers on p2p storage rely on pair-wise contracts for policies that reward users for their participation, all the practical implementations we are aware of mix peers' storage space to create an illusion of a single "disk".

In our system, each user is responsible for keeping as many replication contracts as she needs. To locate replicas, a user keeps a catalog of her data that specifies which peers store which data chunks. As placement of the replicas is explicitly present, in contrast to storing the data in a DHT, the placement can be optimized to a specific network topology, which allow to take into account e.g. geographical dispersion of the nodes.

A common problem in p2p systems are transient failures of peers. In order to propagate updates when (some of) the replicas are off-line, we have proposed a mechanism of asynchronous messages. In the experimental evaluation, we

confirmed the positive influence of asynchronous messaging and synchro-peers on the message delivery time and on probability that an update is successful. With 5 synchro-peers the receiver average waiting time for a message delivery drops from 5.8h to 0.9h; the probability of successful delivery of the message to any synchro-peer increases from 0.46 to 0.9. As being a synchro-peer does not significantly increase peer's load (because synchro-peers only rely control messages, not the data), synchro-peers are an inexpensive way to increase reliability.

We have proposed a distributed algorithm that optimizes replica placement. The algorithm relies on replicas actively looking for poorly replicated data items. Replicas propose replication contracts to owners of such items. We have proposed robust mechanism of changing contracts that can assure the proper load balancing (after 3 days the relative error drops to 8%) even in the environment with low peers' availability (median equal to 13%) and thus high synchronization time (relative to the sender, the average and maximum are equal to 2.45h and 52h, which gives the corresponding absolute times equal to 20h and 289h).

We have implemented a prototype and tested it on 150 computers in our faculty and 50 computers in Planet-Lab. The prototype implementation (with the source code) can be downloaded from: <http://dl.dropbox.com/u/70965741/p2pBackup.tar>; and will be released with an open-source license.

The experiments show that the backup time increases significantly if machines are weakly-available: from 0.76h for nearly always-available Planet-Lab nodes to 3.1h for our lab with just 13% average availability. This *cost of unavailability* makes some environments less suitable for p2p backup. The irregular environments negatively influence the maximal durations of transferring data. But choosing machines with better availability strongly reduces this effect (by restricting our lab environment to machines with more than 20% availability, the average backup time decreases from 3.1h to 1.9h). Moreover, in enterprise environments such irregular availabilities should not be the case. There, however, the machines may have their specific, regular availability patterns. In such case it may be valuable to use more sophisticated availability models.

Yet, we must stress that our back-up system works even on the environment with weakly-available machines having irregular session times.

In our future work, we plan to further develop the prototype. For instance, erasure coding of chunks (instead of complete replication) can be easily added. Our longer-term plan is to evolve the system so that it could constitute a standard data storage layer for higher-level p2p applications, such as Distributed On-line Social Networks (DOSNs), which would require not only incentive-compatible policies, but also collaborative sharing and updating of some of the data items.

REFERENCES

- [1] A. Adya, W. J. Bolosky, M. Castro, G. Cermak, R. Chaiken, J. R. Douceur, J. Howell, J. R. Lorch, M. Theimer, and R. P. Wattenhofer, "Farsite: Federated, available, and reliable storage for an incompletely trusted environment," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 1–14, 2002.
- [2] A. Muthitacharoen, R. Morris, T. M. Gil, and B. Chen, "Ivy: A read/write peer-to-peer file system," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 31–44, 2002.
- [3] J. Michel Busca, F. Picconi, and P. Sens, "Pastis: A highly-scalable multi-user peer-to-peer file system," in *Euro-Par, Proc.*, 2005.
- [4] B. Amann, B. Elser, Y. Hourri, and T. Fuhrmann, "Igorfs: A distributed p2p file system." Los Alamitos, CA, USA: IEEE Computer Society, 2008, pp. 77–78.
- [5] Z. Zhang, Q. Lian, S. Lin, W. Chen, Y. Chen, and C. Jin, "Bitvault: a highly reliable distributed data retention platform," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 27–36, April 2007.
- [6] F. Chang, J. Dean, S. Ghemawat, W. C. Hsieh, D. A. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. E. Gruber, "Bigtable: A distributed storage system for structured data," *ACM Trans. Comput. Syst.*, vol. 26, pp. 4:1–4:26, June 2008.
- [7] K. Rzađca, A. Datta, and S. Buchegger, "Replica placement in p2p storage: Complexity and game theoretic analyses," in *ICDCS 2010, Proc.* IEEE Computer Society, 2010, pp. 599–609.
- [8] L. Cox and B. Noble, "Samsara: Honor among thieves in peer-to-peer storage," in *ACM SOSP, Proc.*, 2003, pp. 120–132.
- [9] S. Ghemawat, H. Gobioff, and S. T. Leung, "The Google file system," in *ACM SIGOPS, Proc.*, vol. 37, no. 5. ACM, 2003, pp. 29–43.
- [10] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki, "Hydrastor: a scalable secondary storage," in *FAST, Proceedings*, 2009, pp. 197–210.
- [11] B. Zhu, K. Li, and H. Patterson, "Avoiding the disk bottleneck in the data domain deduplication file system," in *FAST, Proc.*, 2008, pp. 18:1–18:14.
- [12] D. Geer, "Reducing the storage burden via data deduplication," *Computer*, vol. 41, pp. 15–17, 2008.
- [13] S. Bernard and F. Le Fessant, "Optimizing peer-to-peer backup using lifetime estimations," in *Damap Proc.*, 2009.
- [14] J. Douceur and R. Wattenhofer, "Competitive hill-climbing strategies for replica placement in a distributed file system," in *DISC, Proc.*, ser. LNCS, vol. 2180. Springer, 2001, pp. 48–62.
- [15] B. Chun, F. Dabek, A. Haebleren, E. Sit, H. Weatherspoon, M. Kaashoek, J. Kubiatowicz, and R. Morris, "Efficient replica maintenance for distributed storage systems," in *NSDI, Proc.*, vol. 6, 2006.
- [16] R. Bhagwan, S. Savage, and G. Voelker, "Replication strategies for highly available peer-to-peer storage systems," in *Fu-DiCo, Proc.*, ser. LNCS. Springer, 2002.
- [17] R. Rodrigues and B. Liskov, "High availability in dhfs: Erasure coding vs. replication," in *IPTPS, Proc.*, ser. LNCS, vol. 3640. Springer, 2005.
- [18] F. Dabek, M. F. Kaashoek, D. Karger, R. Morris, and I. Stoica, "Wide-area cooperative storage with cfs," in *SOSP, Proc.*, 2001.
- [19] A. Rowstron and P. Druschel, "Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems," 2001.
- [20] J. Kubiatowicz, D. Bindel, Y. Chen, S. Czerwinski, P. Eaton, D. Geels, R. Gummadi, S. Rhea, H. Weatherspoon, W. Weimer, C. Wells, and B. Zhao, "Oceanstore: an architecture for global-scale persistent storage," *SIGPLAN Not.*, vol. 35, pp. 190–201, November 2000.
- [21] <http://www.cleversafe.com/>.
- [22] <http://www.wuala.com/>.
- [23] <http://sourceforge.net/projects/p2pbackupsmile/>.
- [24] <http://zoogmo.wordpress.com/>.
- [25] <https://launchpad.net/colonyfs>.
- [26] <http://freenetproject.org/>.
- [27] W. J. Bolosky, J. R. Douceur, and J. Howell, "The Farsite project: a retrospective," *SIGOPS Oper. Syst. Rev.*, vol. 41, pp. 17–26, April 2007.
- [28] J. J. Kistler and M. Satyanarayanan, "Disconnected operation in the Coda file system," *ACM Trans. Comput. Syst.*, vol. 10, pp. 3–25, 1992.
- [29] J. R. Douceur and J. Howell, "Byzantine fault isolation in the Farsite distributed file system," in *IPTPS, Proc.*, 2006.
- [30] J. R. Douceur and R. P. Wattenhofer, "Optimizing file availability in a secure serverless distributed file system," in *SRDS, Proc.*, 2001.
- [31] A. Adya, W. J. Bolosky, J. R. Chaiken, J. R. Douceur, J. Howell, and J. Lorch, "Load management in a large-scale decentralized file system," Microsoft Research, Tech. Rep., July 2004.